Semantics of exceptions

K. Rustan M. Leino, Jan L.A. van de Snepscheut
6 October 1993

This note describes a trace semantics of exceptions from which a weakest precondition semantics is derived.

1 Introduction

We describe a trace semantics of exceptions and then derive a weakest precondition semantics. A program that contains exceptions terminates in one of two possible ways (if it terminates at all): either it terminates exceptionally or it terminates normally. We will therefore consider weakest preconditions that are functions of two postconditions. As a preparation we study aribitrary functions of two arguments, and their compositions.

2 Left and right composition of functions on pairs

Functions of type $D \times D \rightarrow D$ for any domain D may be composed in different ways. We use f, g, and h to denote any such function, and p and q to denote any elements in D. An ordered pair with components p and q is written (p, q).

Functions from pairs to elements can be composed in different ways. We first consider *left* and *right* composition, written $\langle\circ$ and $\circ\rangle$, respectively. We define these as follows.

$$(f \langle\circ\, g).(p, q) \;=\; f.(g.(p, q), q) \tag{1}$$

$$(f \circ\rangle\, g).(p, q) \;=\; f.(p, g.(p, q)) \tag{2}$$

Theorem

$$\langle\circ \text{ is associative.} \tag{3}$$

$$\circ\rangle \text{ is associative.} \tag{4}$$

Proof.

$$
\begin{array}{ll}
 & (f \langle\circ (g \langle\circ h)).(p,q) \\
= & \quad \{ \quad (1)\text{: def. of } \langle\circ \quad \} \\
 & f.((g \langle\circ h).(p,q),q) \\
= & \quad \{ \quad (1)\text{: def. of } \langle\circ \quad \} \\
 & f.(g.(h.(p,q),q),q) \\
= & \quad \{ \quad (1)\text{: def. of } \langle\circ \quad \} \\
 & (f \langle\circ g).(h.(p,q),q) \\
= & \quad \{ \quad (1)\text{: def. of } \langle\circ \quad \} \\
 & ((f \langle\circ g) \langle\circ h).(p,q)
\end{array}
$$

We omit the proof of the other case as it is similar to the present case (and we will do so in many more proofs). □

Two functions of special interest are L and R defined as:

$$L.(p,q) \;=\; p \tag{5}$$

$$R.(p,q) \;=\; q \tag{6}$$

Theorem

L is the identity of $\langle\circ$. (7)

R is the identity of $\circ\rangle$. (8)

Proof.

$$
\begin{array}{ll}
 & (L \langle\circ f).(p,q) \\
= & \quad \{ \quad (1)\text{: def. of } \langle\circ \quad \} \\
 & L.(f.(p,q),q) \\
= & \quad \{ \quad (5)\text{: def. of } L \quad \} \\
 & f.(p,q) \\
= & \quad \{ \quad (5)\text{: def. of } L \quad \} \\
 & f.(L.(p,q),q) \\
= & \quad \{ \quad (1)\text{: def. of } \langle\circ \quad \} \\
 & (f \langle\circ L).(p,q)
\end{array}
$$

□

Theorem

$$L \text{ is a left zero of } \circ\rangle \,. \tag{9}$$

$$R \text{ is a left zero of } \langle\circ \,. \tag{10}$$

Proof.

$$\begin{array}{ll} & (L \circ\rangle\, g).(p,q) \\ = & \quad \{ \quad (2)\text{: def. of } \circ\rangle \quad \} \\ & L.(p, g.(p,q)) \\ = & \quad \{ \quad (5)\text{: def. of } L \quad \} \\ & p \\ = & \quad \{ \quad (5)\text{: def. of } L \quad \} \\ & L.(p,q) \end{array}$$

□

3 Double composition

We define *double* composition, written $\langle\circ\rangle$.

$$(f \,\langle\circ\rangle\, g).(p,q) \;=\; f.(g.(p,q), g.(p,q)) \tag{11}$$

We have the following correspondences between single (left and right) compositions and double composition.

Theorem

$$f \,\langle\circ\rangle\, g \;=\; (f \,\langle\circ\, R) \circ\rangle\, g \tag{12}$$

$$f \,\langle\circ\rangle\, g \;=\; (f \circ\rangle\, L) \,\langle\circ\, g \tag{13}$$

Proof. We calculate,

$$\begin{array}{ll} & ((f \,\langle\circ\, R) \circ\rangle\, g).(p,q) \\ = & \quad \{ \quad (2)\text{: def. of } \circ\rangle \quad \} \\ & (f \,\langle\circ\, R).(p, g.(p,q)) \\ = & \quad \{ \quad (1)\text{: def. of } \langle\circ \quad \} \\ & f.(R.(p, g.(p,q)), g.(p,q)) \end{array}$$

$$
\begin{array}{ll}
= & \quad \{ \quad (6)\text{: def. of } R \quad \} \\
& f.(g.(p,q), g.(p,q)) \\
= & \quad \{ \quad (11)\text{: def. of } \langle\circ\rangle \quad \} \\
& (f \mathbin{\langle\circ\rangle} g).(p,q)
\end{array}
$$

□

We continue with some theorems regarding the associativity and distributivity of the composition operators.

Theorem

$$(f \mathbin{\langle\circ\rangle} g) \mathbin{\langle\circ} h \;=\; f \mathbin{\langle\circ\rangle} (g \mathbin{\langle\circ} h) \tag{14}$$

$$(f \mathbin{\langle\circ\rangle} g) \mathbin{\circ\rangle} h \;=\; f \mathbin{\langle\circ\rangle} (g \mathbin{\circ\rangle} h) \tag{15}$$

Proof.

$$
\begin{array}{ll}
& (f \mathbin{\langle\circ\rangle} g) \mathbin{\langle\circ} h \\
= & \quad \{ \quad (13)\text{: double/single trade; (3): } \langle\circ \text{ is associative} \quad \} \\
& (f \mathbin{\circ\rangle} L) \mathbin{\langle\circ} g \mathbin{\langle\circ} h \\
= & \quad \{ \quad (13)\text{: double/single trade; (3): } \langle\circ \text{ is associative} \quad \} \\
& f \mathbin{\langle\circ\rangle} (g \mathbin{\langle\circ} h)
\end{array}
$$

□

Theorem

$\langle\circ\rangle$ is associative. (16)

Proof.

$$
\begin{array}{ll}
& f \mathbin{\langle\circ\rangle} (g \mathbin{\langle\circ\rangle} h) \\
= & \quad \{ \quad (12)\text{: double/single trade } \} \\
& f \mathbin{\langle\circ\rangle} ((g \mathbin{\langle\circ} R) \mathbin{\circ\rangle} h) \\
= & \quad \{ \quad (15)\text{: associativity of } \langle\circ\rangle \quad \circ\rangle \quad \} \\
& (f \mathbin{\langle\circ\rangle} (g \mathbin{\langle\circ} R)) \mathbin{\circ\rangle} h \\
= & \quad \{ \quad (14)\text{: associativity of } \langle\circ\rangle \quad \langle\circ \quad \} \\
& ((f \mathbin{\langle\circ\rangle} g) \mathbin{\langle\circ} R) \mathbin{\circ\rangle} h \\
= & \quad \{ \quad (12)\text{: double/single trade} \quad \} \\
& (f \mathbin{\langle\circ\rangle} g) \mathbin{\langle\circ\rangle} h
\end{array}
$$

□

Theorem

L and R are left identities of $\langle\circ\rangle$. (17)

Proof.

$$
\begin{array}{ll}
 & L \langle\circ\rangle g \\
= & \quad \{ \quad (12)\text{: double/single trade} \quad \} \\
 & (L \langle\circ R) \circ\rangle g \\
= & \quad \{ \quad (7)\text{: } L \text{ is identity of } \langle\circ \quad \} \\
 & R \circ\rangle g \\
= & \quad \{ \quad (8)\text{: } R \text{ is identity of } \circ\rangle \quad \} \\
 & g \\
= & \quad \{ \quad (7)\text{: } L \text{ is identity of } \langle\circ \quad \} \\
 & L \langle\circ g \\
= & \quad \{ \quad (8)\text{: } R \text{ is identity of } \circ\rangle \quad \} \\
 & (R \circ\rangle L) \langle\circ g \\
= & \quad \{ \quad (13)\text{: double/single trade} \quad \} \\
 & R \langle\circ\rangle g
\end{array}
$$

□

A consequence of this theorem, since L and R differ, is that $\langle\circ\rangle$ lacks a right identity. However, $\langle\circ\rangle$ with L or R as a second argument is still interesting, as is shown by the next theorem.

Theorem

$$f \langle\circ\rangle L \;=\; f \circ\rangle L \tag{18}$$

$$f \langle\circ\rangle R \;=\; f \langle\circ R \tag{19}$$

Proof.

$$
\begin{array}{ll}
 & f \langle\circ\rangle L \\
= & \quad \{ \quad (13)\text{: double/single trade} \quad \} \\
 & (f \circ\rangle L) \langle\circ L \\
= & \quad \{ \quad (7)\text{: } L \text{ is identity of } \langle\circ \quad \} \\
 & f \circ\rangle L
\end{array}
$$

□

As we find these instances where $\langle\circ\rangle$ equals $\circ\rangle$ or $\langle\circ$ interesting, we introduce some special notation, $\lceil\ \rceil$ and $\lfloor\ \rfloor$, defined as follows.

$$\lceil f \rceil \;=\; f \,\langle\circ\rangle\, L \tag{20}$$

$$\lfloor f \rfloor \;=\; f \,\langle\circ\rangle\, R \tag{21}$$

This leads us to the following theorem.

Theorem

$$\lceil L \rceil = \; L \; = \lceil R \rceil \tag{22}$$

$$\lfloor L \rfloor = \; R \; = \lfloor R \rfloor \tag{23}$$

Proof.

$$\begin{array}{ll}
 & \lceil L \rceil \\
= & \qquad \{ \quad (20)\text{: def. of } \lceil\ \rceil \quad \} \\
 & L \,\langle\circ\rangle\, L \\
= & \qquad \{ \quad (18) \quad \} \\
 & L \circ\rangle\, L \\
= & \qquad \{ \quad (9)\text{: } L \text{ is left zero of } \circ\rangle \quad \} \\
 & L \\
= & \qquad \{ \quad (8)\text{: } R \text{ is identity of } \circ\rangle \quad \} \\
 & R \circ\rangle\, L \\
= & \qquad \{ \quad (18) \quad \} \\
 & R \,\langle\circ\rangle\, L \\
= & \qquad \{ \quad (20)\text{: def. of } \lceil\ \rceil \quad \} \\
 & \lceil R \rceil
\end{array}$$

□

Theorem

$\lceil\ \rceil$ and $\lfloor\ \rfloor$ are idempotent. (24)

Proof.

$\lfloor \lfloor f \rfloor \rfloor$

$=$ { (21): definition of $\lfloor\ \rfloor$; (16): $\langle\circ\rangle$ is associative }

$f \langle\circ\rangle R \langle\circ\rangle R$

$=$ { (17): R is left identity of $\langle\circ\rangle$ }

$f \langle\circ\rangle R$

$=$ { (21): definition of $\lfloor\ \rfloor$ }

$\lfloor f \rfloor$ □

Theorem

$$f \langle\circ\rangle g \ = \ \lceil f \rceil \langle\circ\, g \tag{25}$$

$$f \langle\circ\rangle g \ = \ \lfloor f \rfloor \circ\rangle\, g \tag{26}$$

Proof. Immediate from (12),(13) and (19),(18) and (21),(20). □

Theorem

$$\lceil f \langle\circ\rangle g \rceil \ = \ \lceil f \rceil \langle\circ\rangle \lceil g \rceil \tag{27}$$

$$\lfloor f \langle\circ\rangle g \rfloor \ = \ \lfloor f \rfloor \langle\circ\rangle \lfloor g \rfloor \tag{28}$$

Proof.

$\lfloor f \langle\circ\rangle g \rfloor$

$=$ { (21): definition of $\lfloor\ \rfloor$; (16): $\langle\circ\rangle$ is associative }

$f \langle\circ\rangle g \langle\circ\rangle R$

$=$ { (17): R is left identity of $\langle\circ\rangle$; (16): $\langle\circ\rangle$ is associative }

$f \langle\circ\rangle R \langle\circ\rangle g \langle\circ\rangle R$

$=$ { (21): definition of $\lfloor\ \rfloor$; (16): $\langle\circ\rangle$ is associative }

$\lfloor f \rfloor \langle\circ\rangle \lfloor g \rfloor$ □

Theorem

$$\lceil f \langle\circ\rangle g \rceil \ = \ \lceil f \rceil \langle\circ\, \lceil g \rceil \tag{29}$$

$$\lfloor f \langle\circ\rangle g \rfloor \ = \ \lfloor f \rfloor \circ\rangle \lfloor g \rfloor \tag{30}$$

Proof. Immediate from (27), (28) and (25), (26) and (24). □

4 Transposition

We introduce operator $\sim$ with higher binding power than composition and function application, defined as follows.

$$\sim f.(p,q) \;=\; f.(q,p) \tag{31}$$

Clearly, $\sim$ is an involution, that is, $\sim\sim$ is the identity function.

Theorem

$$\sim (f \mathbin{\langle\circ} g) \;= \sim f \mathbin{\circ\rangle} \sim g \tag{32}$$

$$\sim (f \mathbin{\circ\rangle} g) \;= \sim f \mathbin{\langle\circ} \sim g \tag{33}$$

$$\sim (f \mathbin{\langle\circ\rangle} g) \;= \sim f \mathbin{\langle\circ\rangle} \sim g \tag{34}$$

Proof. For left composition, we have

$$\begin{array}{ll} & \sim (f \mathbin{\langle\circ} g).(p,q) \\ = & \quad \{ \;\; (31)\text{: definition of } \sim \;\; \} \\ & (f \mathbin{\langle\circ} g).(q,p) \\ = & \quad \{ \;\; (1)\text{: definition of } \langle\circ \;\; \} \\ & f.(g.(q,p),p) \\ = & \quad \{ \;\; (31)\text{: definition of } \sim \;\; \} \\ & f.(\sim g.(p,q),p) \\ = & \quad \{ \;\; (31)\text{: definition of } \sim \;\; \} \\ & \sim f.(p,\sim g.(p,q)) \\ = & \quad \{ \;\; (2)\text{: definition of } \circ\rangle \;\; \} \\ & (\sim f \mathbin{\circ\rangle} \sim g).(p,q) \qquad , \end{array}$$

and similar for right composition. For double composition, we have

$$\begin{array}{ll} & \sim (f \mathbin{\langle\circ\rangle} g).(p,q) \\ = & \quad \{ \;\; (31)\text{: definition of } \sim \;\; \} \\ & (f \mathbin{\langle\circ\rangle} g).(q,p) \\ = & \quad \{ \;\; (11)\text{: definition of } \langle\circ\rangle \;\; \} \end{array}$$

$$
\begin{array}{ll}
 & f.(g.(q,p), g.(q,p)) \\
= & \quad \{ \ (31)\text{: definition of } \sim, \text{ twice } \} \\
 & f.(\sim g.(p,q), \sim g.(p,q)) \\
= & \quad \{ \ (31)\text{: definition of } \sim \ \} \\
 & \sim f.(\sim g.(p,q), \sim g.(p,q)) \\
= & \quad \{ \ (11)\text{: definition of } \langle\circ\rangle \ \} \\
 & (\sim f \langle\circ\rangle \sim g).(p,q) \qquad .
\end{array}
$$

□

This theorem shows the duality between $\langle\circ$ and $\circ\rangle$.

5 Trace semantics

In this section we turn to programs that can raise and handle exceptions. We follow the path of [4] and [3] which describe first an operational semantics in terms of traces and then derives a weakest precondition semantics from it. Our programs operate on a state that includes one coordinate per program variable plus one coordinate, oc, for indicating whether the outcome is normal or exceptional. For state x, we write $x.oc = nor$ to indicate that the outcome is normal, and $x.oc = exc$ to indicate that the outcome is exceptional. We write X for the set of all states, including those with an exceptional outcome. The semantics of a program is defined via traces. In this note, a trace is a nonempty sequence of states; no actions are recorded in the traces. Every trace starts with a normal state. A trace set is a (possibly infinite) set of (possibly infinite) traces. For program S, we identify S with the set of all traces that can be the result of executing S. For every state x, set S has at least one trace starting with x if $x.oc = nor$ and no trace starting with x if $x.oc = exc$. Catenation will be denoted by juxtaposition. Variables s and t range over (possibly empty) sequences of states, and x and y over states.

By way of introduction, we define the trace semantics of the assignment statement. In the absence of exceptions one would write

$$v := E \ = \ \{x :: x \ x[v := E.x]\}$$

in which every trace has length two: it consists of initial state x and final state $x[v := E.x]$, that is, state x in which the value of coordinate v has been replaced by the the value of expression E evaluated in state x. Set $v := E$ contains such a trace for every state $x \in X$. In the presence of exceptions, we restrict x to be a normal state and write

$$v := E \ = \ \{x : x.oc = nor : x \ x[v := E.x]\} \qquad . \tag{35}$$

Statement *skip* is defined as

$$skip \ = \ \{x : x.oc = nor : x\} \tag{36}$$

in which the latter occurrence of x denotes a trace of length one. We could have chosen

$$skip \ = \ \{x : x.oc = nor : x \ x\}$$

and get traces of length two but, for reasons discussed below, we prefer (36).

We write the raising of an exception as the statement *raise* and we define its trace semantics as

$$raise \;=\; \{x : x.oc = nor : x \; x[oc := exc]\} \tag{37}$$

that is, the set of all traces of length two starting with a normal state and ending with an exceptional state; the two states are equal in every other coordinate. Alternatively, we might write

$$raise \;=\; oc := exc$$

except that oc is not a regular program variable; it is a variable that we have introduced for describing the trace semantics only.

The definition of sequential composition is changed to accomodate exceptional outcomes. If there were no exceptions, we could define

$$S;\, T \;=\; \{s, x, t : sx \in S \wedge xt \in T \wedge |s| < \infty : sxt\} \cup \{s : s \in S \wedge |s| = \infty : s\}$$

which distinguishes between those traces in which execution of S does or does not terminate. In the presence of exceptions, we refine the definition to

$$\begin{aligned} S;\, T \;=\; & \{s, x, t : sx \in S \wedge xt \in T \wedge |s| < \infty \wedge x.oc = nor : sxt\} \cup \\ & \{s : s \in S \wedge (|s| = \infty \vee last.s.oc = exc) : s\} \end{aligned} \tag{38}$$

which captures the fact that execution of $S;\, T$ reduces to execution of S in the case where that execution terminates exceptionally. We have

Theorem

$$;\ \text{is associative} \tag{39}$$

Theorem

$$skip \ \text{is the left identity of} \ ; \tag{40}$$

Proof.

$$\begin{aligned} & skip;\, T \\ = \quad & \{\ (38)\text{: def. of } ;\ \} \\ & \{s, x, t : sx \in skip \wedge xt \in T \wedge |s| < \infty \wedge x.oc = nor : sxt\} \cup \\ & \{s : s \in skip \wedge (|s| = \infty \vee last.s.oc = exc) : s\} \end{aligned}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ (36): def. of } \mathit{skip} \ \} \\
& \{x, t : xt \in T \wedge x.oc = nor : xt\} \\
= & \quad \{ \text{ all traces begin in a normal state } \} \\
& T \qquad \Box
\end{array}
$$

Theorem

skip is the right identity of ; (41)

Proof.

$$
\begin{array}{ll}
& S; \mathit{skip} \\
= & \quad \{ \text{ (38): def. of } ; \ \} \\
& \{s, x, t : sx \in S \wedge xt \in \mathit{skip} \wedge |s| < \infty \wedge x.oc = nor : sxt\} \cup \\
& \{s : s \in S \wedge (|s| = \infty \vee \mathit{last}.s.oc = exc) : s\} \\
= & \quad \{ \text{ (36): def. of } \mathit{skip} \ \} \\
& \{s, x : sx \in S \wedge |s| < \infty \wedge x.oc = nor : sx\} \cup \{s : s \in S \wedge (|s| = \infty \vee \mathit{last}.s.oc = exc) : s\} \\
= & \\
& \{s : s \in S : s\} \\
= & \\
& S \qquad \Box
\end{array}
$$

For the above two theorems to hold, it is essential that skip does not duplicate the state in a trace when joined by a semicolon with another statement. This is why the trace set of skip contains traces of length one instead of traces of length two.

Next, we define the trace semantics of the exception handler. We write $S \lhd T$ for the statement whose execution consists of executing S and, if execution thereof does not terminate or terminates normally then no further action is taken; if, however, execution of S terminates exceptionally, then T is executed.

$$
\begin{array}{ll}
S \lhd T \ = & \{s, x, t : sx \in S \wedge x[oc := nor]t \in T \wedge |s| < \infty \wedge x.oc = exc : sxt\} \cup \\
& \quad \{s : s \in S \wedge (|s| = \infty \vee \mathit{last}.s.oc = nor) : s\}
\end{array} \qquad (42)
$$

It would be nice to have

raise is the left and right identity of $\lhd$

but neither part of this property holds because the traces of *raise* have length two, and therefore add an extra state to the traces of the exception handler.

The definitions of *abort* and of the if-statement need not be changed (compared to [4] and [3]) because the initial state is always normal. The definition of the do-statement need not be changed because it is defined in terms of the if-statement, sequential composition, and *skip*. The latter two have already been redefined to cater for exceptional states and the only properties used in the context of the do-statement is that sequential composition is associative, which it still is, and that *skip* is its identity element, which it still is.

6 Weakest preconditions

We define function $wep.S.(P, Q)$ to be the weakest condition on the initial state such that execution of program S terminates; every exceptional outcome satisfies P and every normal outcome satisfies Q. Since the oc coordinate is not part of the program but of the trace semantics only, we require that P, Q, and $wep.S.(P, Q)$ be independent of the oc coordinate, that is, $Q.x = Q.(x[oc := exc]) = Q.(x[oc := nor])$. We do so by restricting P and Q to predicates in which oc does not occur, and by not introducing oc in wep. As a result, we have $wp.S.Q = wep.S.(false, Q)$ as a link between Dijkstra's weakest precondition wp (cf. [2]) and our wep.

In the sequel, we often need to distinguish between conditions on the exceptional and on the normal states, and we write a pair of conditions to capture this distinction. We write

$$(P, Q).x \;=\; (x.oc = exc \wedge P.x) \;\vee\; (x.oc = nor \wedge Q.x) \tag{43}$$

The definition of $wep.S.(P, Q)$ for normal state x is a condition on x such that every trace t of S that begins with initial state x is of finite length and satisfies $(P, Q).(last.t)$. The condition that t be a trace of S and begin with normal state x is concisely coded as $t \in \{x\}; S$ in which $\{x\}$ is a set containing one trace; it is of length one and its only state is x.

$$wep.S.(P, Q).x \;=\; \forall(t : t \in \{x\}; S : |t| < \infty \;\wedge\; (P, Q).(last.t)) \tag{44}$$

We calculate wep for the various program constructs. First, we look at *skip*. For every normal state x,

$$\begin{array}{ll}
 & wep.skip.(P, Q).x \\
= & \quad \{\ (44)\text{: def. of } wep\ \} \\
 & \forall(t : t \in \{x\}; skip : |t| < \infty \;\wedge\; (P, Q).(last.t)) \\
= & \quad \{\ (36)\text{: def. of } skip\ \} \\
 & (P, Q).x \\
= & \quad \{\ x.oc = nor\ \} \\
 & Q.x
\end{array}$$

and hence

$$wep.skip.(P, Q) \;=\; Q \qquad . \tag{45}$$

By a similar calculation we can obtain for every normal state x,

$$\begin{array}{ll}
 & wep.raise.(P, Q).x \\
= & \quad \{\ (44)\text{: def. of } wep\ \} \\
 & \forall(t : t \in \{x\}; raise : |t| < \infty \;\wedge\; (P, Q).(last.t)) \\
= & \quad \{\ (37)\text{: def. of } raise\ \} \\
 & (P, Q).x[oc := exc] \\
= & \quad \{\ (43)\text{: def. of } (P, Q)\ \} \\
 & P.x[oc := exc] \\
= & \quad \{\ P \text{ is independent of } oc\ \} \\
 & P.x
\end{array}$$

and hence

$$wep.raise.(P, Q) \;=\; P \qquad . \tag{46}$$

More involved is the calculation for sequential composition.

$$\begin{array}{ll}
 & wep.(S;\ T).(P, Q).x \\
= & \quad \{\ (44)\text{: def. of } wep\ \} \\
 & \forall(t : t \in \{x\}; S; T : |t| < \infty \;\wedge\; (P, Q).(last.t)) \\
= & \quad \{\ (38)\text{: def. of } ;\ \} \\
 & \forall(s, y, t : sy \in \{x\}; S \wedge yt \in T \wedge |s| < \infty \wedge y.oc = nor : |syt| < \infty \wedge (P, Q).(last.syt)) \;\wedge \\
 & \forall(s : s \in \{x\}; S \wedge (|s| = \infty \vee last.s.oc = exc) : |s| < \infty \wedge (P, Q).(last.s)) \\
= & \quad \{\ \text{calculus; rename } s \text{ and } t \text{ in first quantification}\ \} \\
 & \forall(s : s \in \{x\}; S \wedge |s| < \infty \wedge last.s.oc = nor : \\
 & \quad \forall(t : t \in \{last.s\}; T : |t| < \infty \wedge (P, Q).(last.t))) \;\wedge \\
 & \forall(s : s \in \{x\}; S \wedge (|s| = \infty \vee last.s.oc = exc) : |s| < \infty \wedge (P, Q).(last.s)) \\
= & \quad \{\ \text{calculus; } last.s.oc = exc \;\Rightarrow\; (P, Q).(last.s) = P.(last.s)\ \} \\
 & \forall(s : s \in \{x\}; S : |s| < \infty \;\wedge \\
 & \quad ((last.s.oc = exc \wedge P.(last.s)) \;\vee \\
 & \quad (last.s.oc = nor \wedge \forall(t : t \in \{last.s\}; T : |t| < \infty \wedge (P, Q).(last.t))))
\end{array}$$

$$
\begin{array}{ll}
= & \{\ (44)\text{: def. of } wep\ \} \\
& \forall(s : s \in \{x\};\ S : |s| < \infty\ \wedge \\
& \quad ((last.s.oc = exc \wedge P.(last.s))\ \vee\ (last.s.oc = nor \wedge wep.T.(P, Q).(last.s))) \\
= & \{\ (43)\text{: def. of } (P, Q)\ \} \\
& \forall(s : s \in \{x\};\ S : |s| < \infty \wedge (P, wep.T.(P, Q)).(last.s)) \\
= & \{\ (44)\text{: def. of } wep\ \} \\
& wep.S.(P, wep.T.(P, Q)).x
\end{array}
$$

and hence

$$wep.(S;\ T).(P, Q)\ =\ wep.S.(P, wep.T.(P, Q)) \qquad . \tag{47}$$

Finally we look at exceptional composition

$$
\begin{array}{ll}
& wep.(S \lhd T).(P, Q).x \\
= & \{\ (44)\text{: def. of } wep\ \} \\
& \forall(t : t \in \{x\};\ (S \lhd T) : |t| < \infty\ \ \wedge\ \ (P, Q).(last.t)) \\
= & \{\ (42)\text{: def. of } \lhd\ \} \\
& \forall(s, y, t : sy \in \{x\};\ S \wedge y[oc := nor]t \in T \wedge |s| < \infty \wedge y.oc = exc : |syt| < \infty \wedge (P, Q).(last.syt))\ \wedge \\
& \forall(s : s \in \{x\};\ S \wedge (|s| = \infty \vee last.s.oc = nor) : |s| < \infty \wedge (P, Q).(last.s)) \\
= & \{\ \text{calculus; rename } s \text{ and } t \text{ in first quantification}\ \} \\
& \forall(s : s \in \{x\};\ S \wedge |s| < \infty \wedge last.s.oc = exc : \\
& \quad \forall(t : t \in \{(last.s)[oc := nor]\};\ T : |t| < \infty \wedge (P, Q).(last.t))\ \wedge \\
& \forall(s : s \in \{x\};\ S \wedge (|s| = \infty \vee last.s.oc = nor) : |s| < \infty \wedge (P, Q).(last.s)) \\
= & \{\ \text{calculus; } last.s.oc = nor\ \Rightarrow\ (P, Q).(last.s) = Q.(last.s)\ \} \\
& \forall(s : s \in \{x\};\ S : |s| < \infty\ \wedge \\
& \quad ((last.s.oc = nor \wedge Q.(last.s))\ \vee \\
& \quad (last.s.oc = exc \wedge \forall(t : t \in \{(last.s)[oc := nor]\};\ T : |t| < \infty \wedge (P, Q).(last.t)))) \\
= & \{\ (44)\text{: def. of } wep\ \} \\
& \forall(s : s \in \{x\};\ S : |s| < \infty\ \wedge \\
& \quad ((last.s.oc = nor \wedge Q.(last.s))\ \ \vee\ \ (last.s.oc = exc \wedge (wep.T.(P, Q)).((last.s)[oc := nor])))) \\
= & \{\ (wep.T.(P, Q), Q)\ \text{is independent of } oc\ \} \\
& \forall(s : s \in \{x\};\ S : |s| < \infty\ \wedge \\
& \quad ((last.s.oc = nor \wedge Q.(last.s))\ \ \vee\ \ (last.s.oc = exc \wedge (wep.T.(P, Q)).(last.s)))
\end{array}
$$

$$
\begin{array}{ll}
= & \{\ (43)\text{: def. of } (P, Q)\ \} \\
 & \forall(s : s \in \{x\};\ S : |s| < \infty \wedge (wep.T.(P, Q), Q).(last.s)) \\
= & \{\ (44)\text{: def. of } wep\ \} \\
 & wep.S.(wep.T.(P, Q), Q).x
\end{array}
$$

and obtain

$$wep.(S \lhd T).(P, Q) \ = \ wep.S.(wep.T.(P, Q), Q) \quad . \tag{48}$$

We conclude this section by identifying a program with its *wep*. We then have

from (6) and (45):	*skip*	$=$	R				
from (5) and (46):	*raise*	$=$	L				
from (2) and (47):	$S;\ T$	$=$	$S \circ\rangle\ T$	or, more succinctly,	;	$=$	$\circ\rangle$
from (1) and (48):	$S \lhd T$	$=$	$S \langle\circ\ T$	or, more succinctly,	$\lhd$	$=$	$\langle\circ$

We proceed to show the other operators in this setting. From the definitions of $\lceil\ \rceil$ and $\lfloor\ \rfloor$, and from (18) and (19), we have:

$$\lfloor S \rfloor \ = \ S \lhd \mathit{skip} \quad , \tag{49}$$

$$\lceil S \rceil \ = \ S;\ \mathit{raise} \quad . \tag{50}$$

Using (25) and (26), we have:

$$S \langle\circ\rangle\ T \ = \ (S \lhd \mathit{skip});\ T \quad , \tag{51}$$

$$S \langle\circ\rangle\ T \ = \ (S;\ \mathit{raise}) \lhd T \quad . \tag{52}$$

In words, $\lfloor S \rfloor$ executes S and, provided S terminates, it terminates normally. Similarly, $\lceil S \rceil$ executes S and, provided S terminates, it terminates exceptionally. $S \langle\circ\rangle\ T$ executes S and upon termination —exceptional or normal— executes T.

Transposition $\sim S$ is the statement that terminates just when S does, and upon termination inverts *oc*. We can implement $\sim S$ as

$$
\begin{array}{l}
|[\ \textbf{var}\ \ b : \mathit{boolean}; \\
\quad (S;\ b := \mathit{true}) \lhd b := \mathit{false}; \\
\quad \textbf{if}\ \ b\ \ \textbf{then}\ \ \mathit{raise}\ \ \textbf{fi} \\
]| \quad .
\end{array}
$$

Modula-3 [7] is an example of a programming language with exceptions. In addition to the $\lhd$ construct, it has a so-called *try finally* statement. Execution of

try S **finally** T **end**

consists of the execution of S followed by the execution of T. If the execution of S terminates exceptionally, then execution of T is followed by reraising the exception. This construct can be captured by

$$(S \lhd (T;\ \mathit{raise}));\ T \qquad .$$

Finally, we remark on the relation between the theory presented herein and existing programming languages. We find that usual programming languages introduce an asymmetry between left and right composition. For example, statements begin their execution in a normal state, $\lhd$ is often much longer to type than $;$, and $\lhd$ may not be as efficient as $;$ (see e.g. [7]). However, the properties presented in this note suggest a more symmetric treatment of $;$ and $\lhd$.

7 A programming method

From the weakest-precondition semantics given for a program notation, one often derives some theorems that are used in reasoning about programs. Ideally, they suggest hints for methodical program construction. An example hereof is the invariance theorem for iterative statements (cf. [2]). We give a theorem that suggests using exception handlers in a way similar to split binary semaphores (cf. [6]). The theorem is in terms of Hoare triples for covering normal termination, and free occurrences of *raise* for exceptional termination. First, we give the definition of free occurrence. Statement *raise* occurs free in

- *raise*
- $S0;\ S1$ just when it occurs free in $S0$ or in $S1$
- $S0 \lhd S1$ just when it occurs free in $S1$
- no other statement

Second, we give the rules for Hoare triples $\{R\}\ S\ \{Q\}$ for all statements S. For all Q, R, we have

$$\begin{aligned}
&\{R\}\ x := E\ \{Q\} &&= (R\ \Rightarrow\ Q[x := E])\\
&\{R\}\ \mathit{skip}\ \{Q\} &&= (R\ \Rightarrow\ Q)\\
&\{R\}\ \mathit{raise}\ \{Q\} &&= (Q\ =\ \mathit{false})\\
&\{R\}\ S0;\ S1\ \{Q\} &&= \exists(M : \{R\}\ S0\ \{M\}\ \ \wedge\ \ \{M\}\ S1\ \{Q\})\\
&\{R\}\ S0 \lhd S1\ \{Q\} &&= \exists(M : \{R\}\ S0\ \{Q\}\ \ \wedge\ \ \{M\}\ S1\ \{Q\}\ \wedge\\
&&&\qquad \text{every free occurrence of } \mathit{raise} \text{ in } S0 \text{ has precondition } M)
\end{aligned}$$

The theorem is as follows.

Theorem

If $\{R\}\ S\ \{Q\}$ and every free occurrence of *raise* in S has precondition P, then

$$R \Rightarrow wep.S.(P, Q) \tag{53}$$

Proof. The proof is by induction over the syntax of S. We give three of the cases.

- The Hoare triple $\{R\}\ skip\ \{Q\}$ is equivalent to $R \Rightarrow Q$. Since *raise* does not occur in *skip*, we need to prove $R \Rightarrow wep.skip.(P, Q)$ for all P and this follows directly from (45) and the given $R \Rightarrow Q$.
- The Hoare triple $\{R\}\ raise\ \{Q\}$ is equivalent to $Q = false$. Since *raise* has precondition R, we need to prove $R \Rightarrow wep.raise.(R, false)$ and this follows directly from (46).
- The Hoare triple $\{R\}\ S0 \lhd S1\ \{Q\}$ is equivalent to $\exists(M : \{R\}\ S0\ \{Q\}\ \wedge\ \{M\}\ S1\ \{Q\})$ plus the fact that every free occurrence of *raise* in $S0$ has precondition M. Hence, by induction, $R \Rightarrow wep.S0.(M, Q)$. The free occurrences of *raise* in $S1$ are also the free occurrences of *raise* in $S0 \lhd S1$; thus, they have precondition P. Hence, by induction, $M \Rightarrow wep.S1.(P, Q)$.

$$\begin{array}{ll} & wep.(S0 \lhd S1).(P, Q) \\ = & \quad \{\ (48)\ \} \\ & wep.S0.(wep.S1.(P, Q), Q) \\ \Leftarrow & \quad \{\ wep.S1.(P, Q) \Leftarrow M \text{ by induction; monotonicity (see below)}\ \} \\ & wep.S0.(M, Q) \\ \Leftarrow & \quad \{\ \text{by induction}\ \} \\ & R \end{array}$$

The other two cases are similar. □

This theorem suggests that, when constructing $S0$ in $S0 \lhd S1$, one should have a precondition M for all *raise* statements that occur free in $S0$; the benefit is then that one may assume that same precondition in the construction of $S1$.

In the proof of the theorem above, we appealed to monotonicity. We state, without proof, that $wep.S.(P, Q)$ is a monotonic function of P as well as of Q. A proof can be given by induction over the syntax of S.

As an example, we conclude with a short program. Although the semantics duscussed in the present paper does not explicitly address do- and if-statements, the example contains both. As pointed out before, our semantics can readily be extended to cover those. We give only the program text, and some relevant predicates.

$$
\begin{array}{l}
(\ i := 0; \\
\quad \textbf{do}\ i \neq M \rightarrow j := 0; \\
\qquad\qquad \textbf{do}\ j \neq N \rightarrow \textbf{if}\ a[i,j] = x \rightarrow \mathit{raise}\ [\!]\ a[i,j] \neq x \rightarrow \mathit{skip}\ \textbf{fi}; \\
\qquad\qquad\qquad\qquad j := j + 1 \\
\qquad\qquad \textbf{od}; \\
\qquad\qquad i := i + 1 \\
\quad \textbf{od}; \\
\quad p := \mathit{false} \\
\lhd \\
\quad p := \mathit{true} \\
)
\end{array}
$$

Using invariants $P0$ for the outer loop and $P1$ for the inner loop

$$P0: \quad \forall(m,n : 0 \leq m < i \ \wedge\ 0 \leq n < N : a[m,n] \neq x) \ \wedge\ 0 \leq i \leq M$$

$$P1: \quad P0 \ \wedge\ \forall(n : 0 \leq n < j : a[i,n] \neq x) \ \wedge\ i \neq M \ \wedge\ 0 \leq j \leq N$$

we can infer $P0 \wedge P1 \wedge a[i,j] = x$ as a precondition for *raise*. This in turn allows us to infer postcondition

$$(p \ =\ \exists(m,n : 0 \leq m < M \ \wedge\ 0 \leq n < N : a[m,n] = x)) \ \wedge\ (p \ \Rightarrow\ a[i,j] = x)$$

as a postcondition of the program.

8 Conclusion

In our first attempt, the trace semantics of a program was a set of traces with at least one trace starting with x for each state x, including the exceptional states. As a result, *skip* was no longer an identity of sequential composition, which we needed in order not to have to reprove the semantics of the if- and do-statements as given in [4] and [3]. In the *wep* semantics, however, *skip* is a left and right identity even when states are duplicated.

In a second attempt, we changed the definition of assignment to act as *skip* if the initial state is exceptional. Although this led to the expected semantics, we had to change the trace semantics of too many statements. Besides the assignment, also all other statements had to be changed.

In a third attempt, we changed the semantics of sequential composition so that it would catenate all traces of the two statements, and we changed the definition of exception handling to truncate all traces until their first exceptional state. But this lead to problems with constructs like $(\mathit{raise};\ \mathit{abort}) \lhd T$.

In this note, we do not distinguish between various exceptions. There is no problem in including various exceptions, it just doesn't contribute anything either. If a program has n distinct exceptions

one can change *oc* from a binary valued coordinate to a coordiante that can assume $n+1$ values: the normal value plus one for each exception. It then becomes necessary to refer to this coordinate in the *wep* predicates. As an alternative, one may consider a *wep* of $n+1$ predicative arguments, one for every *oc* value, and let the predicates themselves be independent of *oc*.

We gratefully acknowledge the contributions made by Robert Harley and Peter Hofstee during various discussions. Also, the feedback from Greg Nelson and from other members of IFIP WG 2.3 is greatly appreciated. Although we had not actually seen the paper, we were well aware of the semantics for exceptions given in [5], which is identical to the axiomatic semantics given in the prersent paper. Our trace semantics serves as a foundation thereof. Another early reference to semantics of exception handling is [1]. This reference provides a good discussion of how exception handling can simplify the structure of certain programs. It describes the semantics of exceptions by giving a predicate transformer for normal termination and a separate predicate transformer for exceptional termination. We use one function that maps a pair into a single predicate.

References

[1] F. Christian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163–174, 1984.

[2] E.W. Dijkstra and C.S. Scholten. *Predicate calculus and program semantics.* Springer-Verlag, 1990.

[3] J.J. Lukkien. *Parallel Program Design and Generalized Weakest Preconditions.* PhD thesis, Groningen University, 1991. Also, Caltech technical report CS TR 90-16.

[4] J.J. Lukkien. An operational semantics for the guarded command language. In R.S.Bird, C.C. Morgan, and J.C.P.Woodcock, editors, *Mathematics of Program Construction*, number 669 in Lecture Notes in Computer Science, pages 233–249. Springer-Verlag, 1993.

[5] M.S. Manasse and C.G. Nelson. Correct Compilation of Control Structures. Technical report, AT&T Bell Laboratories, September 1984.

[6] A.J. Martin and J.L.A. van de Snepscheut. Design of synchronization algorithms. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO ASI Series F: Computers and Systems Sciences*, pages 447–478. Springer-Verlag, 1989.

[7] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice-Hall, 1991.